From Noise to Signal: Precisely Identify Affected Packages of Known Vulnerabilities in npm Ecosystem

Yingyuan Pu
QI-ANXIN Technology Research Institute
puyingyuan01@qianxin.com

Lingyun Ying*
QI-ANXIN Technology Research Institute
yinglingyun@qianxin.com

Yacong Gu
Tsinghua University
Tsinghua University-QI-ANXIN Group JCNS
guyacong@tsinghua.edu.cn

***Abstract*—npm is the largest open-source software ecosystem with over 3 million packages. However, its complex dependencies between packages expose it to significant security threats as many packages directly or indirectly depend on other ones with known vulnerabilities. Timely updating these vulnerable dependencies is a big challenge in software supply chain security, primarily due to the widespread effect of vulnerabilities and the huge cost of fixing them. Recent studies have shown that existing package-level vulnerability-propagation-analysis tools lead to high false positives, while function-level tools are not yet feasible for large-scale analysis in the npm ecosystem.**

In this paper, we propose a novel framework VulTracer, which can precisely and efficiently perform vulnerability propagation analysis at function level. By constructing a rich semantic graph for each package independently and then stitching them together, VulTracer can locate vulnerability propagation paths and identify truly affected packages precisely. Through comparative evaluations, our framework achieves an F1 score of 0.905 in call graph construction and reduces false positives from `npm audit` by 94%. We conducted the largest-to-date function-level vulnerability impact measurement on the entire npm ecosystem, covering 34 million package versions. The results demonstrate that 68.28% of potential impacts identified by package-level analysis are merely noise, as the vulnerable code is unreachable. Furthermore, our findings also uncover that true vulnerability propagation (the signal) is shallow, with impact attenuating significantly within just a few dependency hops. VulTracer provides a practical path to mitigate alert fatigue and enables security efforts to focus on genuine, reachable threats.

I. Introduction

Due to the widespread use of JavaScript, npm has become the largest software ecosystem in the open-source world with over 3 million packages [1], and serving approximately 4.5 trillion requests in 2024, a 70% year-over-year increase [2]. This ecosystem's structure is characterized by a high degree of code reuse, often involving numerous small packages that create deeply nested and intricate dependency chains [3]–[6]. While fostering rapid development, this intricate web also creates a fragile software supply chain. Consequently, a single vulnerability in a foundational upstream package can propagate rapidly through the dependency graph, placing a massive number of downstream applications at risk. Recent research shows that approximately one-quarter of all package versions depend on packages with known vulnerabilities [7]. For instance, *pac-resolver*, a widely used npm package (commonly as a dependency) with three million weekly downloads, has a high-severity remote code execution vulnerability. As a result, over 285,000 public repositories on GitHub may be at risk of attack as all of them depend on the package [8].

To manage these complex risks, the industry has widely adopted Software Composition Analysis (SCA) tools. Modern development practices advocate for integrating these tools directly into Continuous Integration/Continuous Deployment (CI/CD) pipelines. This "shift-left" approach, exemplified by automated services like GitHub's Dependabot [9], aims to provide developers with immediate feedback on vulnerabilities with every code change. The goal is to make security a continuous, automated part of the development workflow, rather than a separate, delayed process. This immediacy is considered crucial for effective and timely remediation.

However, SCA tools report primarily on the presence of a vulnerable dependency, leaving developers with a critical and challenging question: *Am I really affected?* Due to the complexity of the dependency network, for a known vulnerable package, users find it extremely difficult to ascertain whether their applications are truly affected or not. This uncertainty is a core problem, as the subsequent process of updating a dependency is a costly endeavor, involving significant effort to ensure compatibility and fix potential breaking changes [10]. In fact, the difficulty of "updating vulnerable dependencies", a task often initiated by imprecise alerts, is considered one of the top five challenges in software supply chain security [11].

The root of this uncertainty lies in the coarse-grained nature

*Lingyun Ying is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.241902
www.ndss-symposium.org

of current SCA tools, which operate at the package-level. This approach fails to determine whether the vulnerable code is actually *reachable*, not to mention exploitable. Our large-scale evaluation confirms this imprecision is severe. Our analysis suggests that 68.28% of alerts generated by package-level analysis appear to be false positives (noise). This massive "alert fatigue" leads to "patching paralysis," where even with fixes available for over 95% of vulnerable components, 80% of enterprise application dependencies remain unmanaged and outdated for over a year [2], [12].

The logical solution to this imprecision is to adopt a finer-grained function-level analysis. While the ultimate goal is to determine true exploitability, assessing function-level reachability stands as the most critical and practical step to resolve the developer's initial uncertainty. Yet fine-grained analysis is hindered by prohibitive computational cost. Jelly [13], the reference implementation of JAM [14], constructs *file-level* summaries per analysis path rather than *package-level* summaries, forcing it to re-scan each affected project separately when upstream vulnerabilities propagate. While modular strategies like Frankenstein [15] scale for statically-typed languages (e.g., Java) via Class Hierarchy Analysis (CHA), such approaches cannot handle JavaScript's dynamic semantics, such as prototype mutation.

Our experiments show that state-of-the-art tools frequently fail due to out-of-memory errors, achieving only a 37.37% success rate under typical resource limits (RQ2 in Section IV). Because automated analysis within the CI/CD pipeline is the primary pathway for improving vulnerability awareness and remediation speed, this performance gap makes existing tools impractical for modern development workflows and hinders dependency security at scale.

To bridge the gap between the need for precision and the practical demands of modern software engineering, any viable solution must overcome several significant technical challenges: **C1)** Scalability and Efficiency. How can we perform fine-grained, function-level analysis without the crippling overhead of whole-program analysis, making it efficient enough for CI/CD and large-scale studies? This requires a departure from the monolithic, re-analyze-everything model. **C2)** Composability and Context. How can we analyze a package in isolation while retaining enough semantic information to accurately connect its CG with those of its dependencies later? This is particularly difficult in JavaScript and requires a novel way to represent and resolve inter-package calls. **C3)** Precision in a Dynamic Language. How can we statically trace call chains with high accuracy through JavaScript's dynamic features, such as higher-order functions and callbacks, which are pervasive and often foil traditional static analysis?

To tackle C1, the scalability challenge, VulTracer's key insight is that the content of an npm package is immutable once published. This makes each (package, version) pair immutable, enabling stable, package-level summaries for ecosystem-scale reuse, which is an opportunity that prior work has not leveraged. We pre-compute a high-fidelity representation for each package, which we term a Rich Semantic Graph (RSG), and cache the result. This "analyze-once, reuse-many-times" model dramatically reduces the computational burden at the build time. To address C2, the composability challenge, we extract a formal interface contract from each RSG. This contract defines the package's public API and external dependencies, providing the essential prerequisite for composition. For C3, the precision challenge, we leverage CodeQL's advanced inter-procedural data-flow analysis to build RSGs. Finally, VulTracer introduces a compositional synthesis algorithm that rapidly stitches these pre-computed RSGs together according to a project's dependency graph, constructing a precise, on-demand CG.

We conducted a comprehensive set of experiments to evaluate VulTracer and the results demonstrate that our approach is not only sound but highly effective. In comparison, VulTracer's CG construction achieves a state-of-the-art F_1 score of 0.905, significantly outperforming prior tools. Notably, its composable design proves highly scalable, successfully analyzing complex, real-world dependency graphs where traditional whole-program analysis tools fail due to memory limitations. In the critical task of auditing software dependencies for security risks, VulTracer successfully reduces 94% of False Positives (FPs) generated by existing tools such as `npm audit`. Furthermore, we conduct the largest-to-date function-level study of 27 distinct CVEs on the entire npm ecosystem (comprising 3,267,273 unique packages across 34,685,976 distinct versions). Our findings reveal that 68.28% of the potential impacts (traced transitively) identified by package-level dependency analysis are effectively FPs, as the vulnerable code is never actually called (i.e., unreachable). We also systematically explore the root causes of this attenuation, showing it stems primarily from the two key factors of unused dependencies and shallow API usage. Our study of six major CVEs found that, on average, nearly a quarter (22.80%) of direct dependents that declare a vulnerable dependency never actually import any code from it. Overall, our contributions in this paper are as follows.

- We design and implement VulTracer[1], utilizing a "analyze-once, reuse-many-times" model to enable scalable function-level analysis.
- VulTracer achieves a state-of-the-art F_1 score of 0.905, reducing FPs by 94% compared to `npm audit`.
- We conduct the largest-to-date function-level impact study on the entire npm ecosystem (over 34M package versions), revealing that 68.28% of package-level alerts are unreachable noise.

II. Background

A. The npm Dependency Ecosystem

npm is the package manager for Node.js, which manages the largest ecosystem in the open-source world. The ecosystem's structure consists of packages and their dependencies. npm package developers specify dependencies in *package.json*,

[1]A live demo of VulTracer is available at https://tianwen.qianxin.com/npm-vultracer/

listing required packages and their versions. This file specifies package names and version ranges using Semantic Versioning [16]. During installation, npm resolves the full dependency graph. This graph includes both direct and indirect (transitive) dependencies. The resulting graph is often deeply nested and highly interconnected. This structure enables high code reuse but also creates complex vulnerability propagation paths. Our work operates on this logical dependency graph to trace inter-package interactions.

B. Static Analysis Challenges in JavaScript

Module Loading and Resolution. Node.js supports two primary module systems: CommonJS (CJS) and ECMAScript Modules (ESM). Specifically, CJS is the traditional system in Node.js. It uses `require()` to load modules and `module.exports` to define public APIs. The `module.exports` object is mutable, and its properties can be modified at runtime [17]. This practice complicates the static determination of a package's public API. Moreover, the argument to `require()` can also be a dynamically computed expression (e.g., `require(variable)`). This further makes static dependency resolution undecidable in the general case [18]. ESM is the modern standard and uses static `import` and `export` declarations. ESM's static structure is more amenable to analysis. However, interoperability between CJS and ESM still creates challenges. The use of transpilers like Babel [19] introduces additional complexity, as a scalable analysis must handle both module systems to accurately model inter-package connections.

Dynamic Language Features. Several core JavaScript features hinder static control-flow tracing. Specifically, functions are first-class citizens [20], so they can be passed as arguments or returned from other functions. This practice obscures the link between a call site and the code being executed. Another challenge is dynamic property access. Properties can be accessed with computed keys, such as `obj[propName]()` [20]. Statically identifying the target function becomes intractable when `propName` is a variable. Finally, the `this` keyword is context-sensitive. Its value depends on how a function is called, not on where it is defined. Resolving `this` statically is a classic problem in program analysis [21], [22]. These dynamic features make simple syntactic analysis highly imprecise; achieving accuracy therefore requires sophisticated data-flow analysis.

C. Reachability vs. Exploitability in Vulnerability Triage

In vulnerability triage, a critical distinction exists between *reachability* and *exploitability*. This distinction is fundamental to understanding the scope and contribution of our work. Reachability analysis determines whether there is a call path to a vulnerable function. Such a path is a *necessary precondition* for an exploit. If unreachable, it cannot be triggered. In contrast, exploitability analysis is a much stronger condition; it determines whether an attacker can actually trigger the vulnerable function. Exploitability requires more than a reachable path. The attacker must also satisfy specific constraints, such as providing malicious input that propagates along the call path. Proving exploitability requires sophisticated techniques like symbolic execution or taint tracking [23]. These techniques are computationally expensive and generally intractable (if not impossible) at an ecosystem scale. Thus, our work focuses on precisely determining reachability at scale. By filtering out alerts for unreachable vulnerable code, we address the critical problem of alert fatigue. This enables developers to focus on the small subset of dependencies that pose a genuine potential threat.

III. Design

This section details the design of VulTracer, our framework for scalable and precise function-level vulnerability analysis. We first provide a high-level overview of our three-phase methodology, then detail each phase's implementation.

A. Overview of VulTracer

Traditional vulnerability analysis based on whole-program CGs suffers from poor scalability. Analyzing an application and its entire dependency graph from scratch requires monolithic, resource-intensive processes. This method is unsuitable for the rapid feedback requirements of modern CI/CD pipelines or ecosystem-scale studies. VulTracer addresses this challenge through a paradigm shift from monolithic analysis to a composable, pre-computed approach. Our methodology consists of three phases that decompose the massive analysis problem into smaller, independent, and reusable components.

As illustrated in Fig. 1, our methodology begins with RSG generation. During this phase, we analyze each package version in complete isolation to generate an RSG. This pre-computation enables reuse across multiple analyses. The second phase extracts formal interface contracts to enable independent RSGs to be connected later. We derive an interface contract from each RSG that serves as a precise, machine-readable specification of the package's public API and external dependencies. This abstraction enables reasoning about inter-package connections without re-examining dependency source code. The final phase performs on-demand compositional synthesis when analyzing specific applications. VulTracer retrieves pre-computed RSGs and their interface contracts for the target application and all dependencies, then applies a lightweight synthesis algorithm to rapidly construct precise, on-demand CGs for vulnerability tracing. This three-phase methodology systematically resolves the conflict between scalability and precision. The approach is predicated on pre-computing a high-fidelity RSG and its formal interface contract for each package, which provides the foundation for rapid, on-demand synthesis.

B. Intra-Package Rich Semantic Graph Generation

The goal of this phase is to generate a detailed, semantically rich representation for each package, encompassing both its internal function call relationships and the essential information required to define its boundaries. This boundary information, namely its external calls and potential API structure, is a

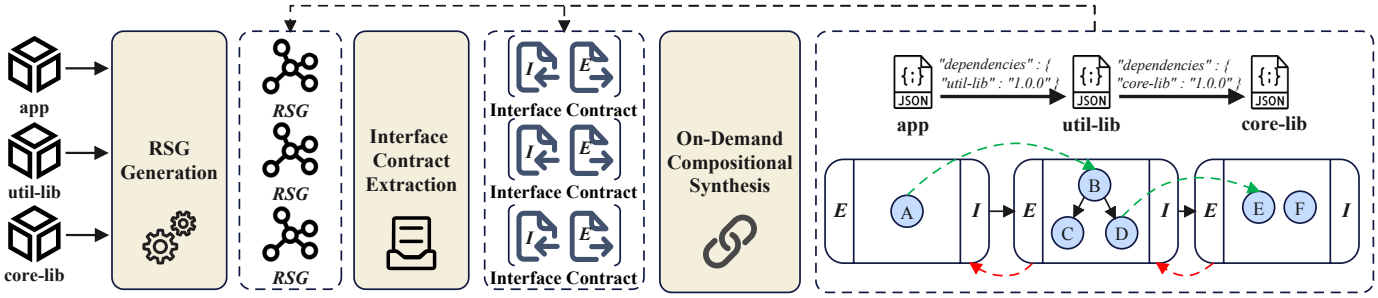

Fig. 1: Overview of VulTracer.

Fig. 2: The three core code snippets of our running example, illustrating the dependency chain from *app* to *util-lib* to *core-lib*. Each snippet highlights a key interaction pattern analyzed by our methodology. The green dashed arrows represent a direct invocation, while red signifies a function reference passed through exports.

```
const utils = require('util-lib');
function handle_result(result) {
  // some processing logic
  return result;
}
function main() {
  res1 = utils.process(data);
  res2 = utils.parser(data);
  res3 = handle_result(res2);
}
main();
```

Listing 1: `app/main.js`

```
const core = require('core-lib');
function process(data) {
  res = core.parse(data);
  return (function(res) {
    // some parsing logic
    return res;
  })(res);
}
module.exports = {
  process: process,
  parser: core.parse // REF-EXPORT
};
```

Listing 2: `util-lib/index.js`

```
function _parse(data) {
  // ... sophisticated logic
  console.log("Core parsing.");
}
module.exports = {
  _parse: _parse
};
```

Listing 3: `core-lib/index.js`

prerequisite for extracting a formal interface contract in the subsequent phase.

The Rich Semantic Graph Design. A standard CG is fundamentally unsuited for composition because it discards the information required to link packages. It loses the semantic context of external calls and fails to model the public API that other packages consume. By omitting these very points of inter-package connection, a CG makes subsequent formalization and composition impossible.

To address limitations of CG, we designed the RSG as a multi-layered model that preserves information discarded by traditional CGs. Specifically, the RSG's key innovation is to reify external calls and API exports into dedicated vertices. This reification creates an explicit model of the package's boundary, which is the prerequisite for building self-contained and composable graphs. Formally, an RSG is a directed graph $G = (V, E)$ where V represents program elements and E represents typed semantic relationships. Def. 1 defines the formal structure of RSG for package P. The vertex set V can be partitioned into three categories: programmatic entities (V_{ent}), invocation points (V_{invk}), and export anchors (V_{export}).

Programmatic entities (V_{ent}) represent static code constructs, partitioned by package boundaries. Module nodes (V_{mod}) correspond to JavaScript modules (also JavaScript files), while function nodes (V_{func}) cover all callable entities including named functions, anonymous functions, class methods, and constructors. Internal modules ($V_{\text{mod\_int}}$) and functions ($V_{\text{func\_int}}$) correspond to constructs defined within P, while external placeholders ($V_{\text{mod\_ext}}$, $V_{\text{func\_ext}}$) represent imported dependencies. These placeholders act as named stubs for the external functions being called, allowing the isolated RSG to be internally consistent before composition. *Invocation points* (V_{invk}) reify call sites as vertices rather than edges, enabling attachment of context-specific information to invocations. These nodes also can be divided into internal invocation nodes ($V_{\text{invk\_int}}$) and external invocation nodes ($V_{\text{invk\_ext}}$). The former represents calls to entities that are resolved within the package boundary, while the latter represents calls to entities outside the package, which are resolved during the compositional synthesis phase. *Export anchors* model the package's public API by reifying export statements as vertices. Each anchor reifies a specific export statement (e.g., an assignment to `module.exports.parser`).

The edge set E is divided into three subsets: lexical-nesting edges (E_{contains}), call-resolution edges (E_{call}), and export-resolution edges (E_{export}). The set of E_{contains} represents edges connect entities based on their syntactic structure, which must be derived from the program's Abstract Syntax Tree (AST), e.g., a module contains a function and a function contains an invocation point. *Resolution Edges* (E_{call}, E_{export}) represents the result of control and data-flow analyses linking an action-oriented vertex (an invocation or an export) to the entity it resolves to. These two semantic-resolution subsets are further partitioned into four subtypes, the first two of which handling function invocations. For internal calls ($E_{\text{int\_call}}$), this edges link an internal invocation to an internal entity ($V_{\text{mod\_int}}$, $V_{\text{func\_int}}$). In contrast, external calls ($E_{\text{ext\_call}}$) link an invocation to an external entity placeholder ($V_{\text{mod\_ext}}$,

$V_{\text{func\_ext}}$). In addition, the other two subtypes model export statements. A standard export ($E_{\text{std\_exp}}$) links an export anchor to an internal entity. A reference export ($E_{\text{ref\_exp}}$) links an export anchor to an external entity placeholder. This models cases where a value from a dependency is passed through the package and exported without being invoked locally. Our running example in `util-lib` (Listing 2 of Fig. 2) illustrates both patterns. The statement in line 10 represents a standard export ($E_{\text{std\_exp}}$), linking the `process` export anchor to the locally defined function `process` (a $V_{\text{func\_int}}$ node). In contrast, line 11 (highlighted in red) represents a reference export ($E_{\text{ref\_exp}}$). It links the `parser` export anchor to an external entity placeholder representing the imported entity `parse` from `core-lib`, and this relationship is illustrated by the red dashed arrow in Fig. 2. This design ensures RSGs are self-contained models encoding both import requirements and export capabilities, enabling subsequent interface contract extraction from V_{export} and $V_{\text{invk\_ext}}$.

DEF 1: Formal structure of RSG

$G = (V, E)$ where for a given package P :
The vertex set: $V \triangleq V_{\text{ent}} \cup V_{\text{invk}} \cup V_{\text{export}}$

- $V_{\text{ent}} \triangleq V_{\text{mod}} \cup V_{\text{func}}$
  - $V_{\text{mod}} \triangleq V_{\text{mod\_int}} \cup V_{\text{mod\_ext}}$, where:
    - $V_{\text{mod\_int}} \triangleq \{v \mid v \text{ is a module defined in } P\}$
    - $V_{\text{mod\_ext}} \triangleq \{v \mid v \text{ is a module imported by } P\}$
  - $V_{\text{func}} \triangleq V_{\text{func\_int}} \cup V_{\text{func\_ext}}$, where:
    - $V_{\text{func\_int}} \triangleq \{v \mid v \text{ is a function defined in } P\}$
    - $V_{\text{func\_ext}} \triangleq \{v \mid v \text{ is a function imported by } P\}$
- $V_{\text{invk}} \triangleq V_{\text{invk\_int}} \cup V_{\text{invk\_ext}}$, where:
  - $\rho : V_{\text{invk}} \rightarrow V_{\text{ent}}$
  - $V_{\text{invk\_int}} \triangleq \{v \in V_{\text{invk}} \mid \rho(v) \in V_{\text{mod\_int}} \cup V_{\text{func\_int}}\}$
  - $V_{\text{invk\_ext}} \triangleq \{v \in V_{\text{invk}} \mid \rho(v) \in V_{\text{mod\_ext}} \cup V_{\text{func\_ext}}\}$
- $V_{\text{export}} \triangleq \{v \mid v \text{ is an export anchor in } P\}$

The edge set: $E \triangleq E_{\text{call}} \cup E_{\text{export}} \cup E_{\text{contains}}$

- $E_{\text{call}} \triangleq \{(v, \rho(v)) \mid v \in V_{\text{invk}}\}$
  - $E_{\text{int\_call}} \triangleq \{(v, \rho(v)) \in E_{\text{call}} \mid \rho(v) \in V_{\text{mod\_int}} \cup V_{\text{func\_int}}\}$
  - $E_{\text{ext\_call}} \triangleq \{(v, \rho(v)) \in E_{\text{call}} \mid \rho(v) \in V_{\text{mod\_ext}} \cup V_{\text{func\_ext}}\}$
- $E_{\text{export}} \triangleq E_{\text{std\_exp}} \cup E_{\text{ref\_exp}}$
  - $E_{\text{std\_exp}} \subseteq V_{\text{export}} \times (V_{\text{mod\_int}} \cup V_{\text{func\_int}})$
  - $E_{\text{ref\_exp}} \subseteq V_{\text{export}} \times (V_{\text{mod\_ext}} \cup V_{\text{func\_ext}})$
- $E_{\text{contains}} \triangleq \{(v, u) \mid v \in V_{\text{ent}}, u \in V, \mathit{contains}(v, u)\}$
- $\mathit{contains} \subseteq V \times V$, denotes the lexical-nesting relation in AST.

RSG Implementations. We implement RSG generation through a systematic pipeline using CodeQL queries over the `javascript-all` library. The process constructs concrete graph instances from package's raw source code.

First, we instantiate the vertex set V by identifying all necessary program elements. Internal entities ($V_{\text{mod\_int}}$, $V_{\text{func\_int}}$) are created by identifying all modules and callable constructs within the package. To ensure the graph is self-contained, placeholder nodes for external entities ($V_{\text{mod\_ext}}$, $V_{\text{func\_ext}}$) are created on-demand for each imported and invoked point. Concurrently, we create the action-oriented nodes. We query for all function call expressions or module import statements to create the invocation nodes in V_{invk}. Each assignments to `module.exports` or `exports` leads to the creation of an export anchor nodes in V_{export}. These nodes are then further abstracted into an interface contract in the next phase.

Following vertex instantiation, we establish the edge set E. Lexical-nesting edges (E_{contains}) are straightforwardly derived from the parent-child relationships in the AST. In contrast, establishing call-resolution edges E_{call} for invocation points V_{invk} is a more involved process. Our primary goal is to resolve each invocation to a concrete callable entity, which determines whether an $E_{\text{int\_call}}$ or an $E_{\text{ext\_call}}$ edge is created. To achieve this, we first attempt to resolve every invocation point within the local package scope. We employ a dual strategy to handle JavaScript's dynamic nature.

For simple, direct calls, we leverage CodeQL's standard value-tracking primitives (e.g., getACallee) to directly identify the callee within the package. For complex cases involving higher-order functions or callbacks, we apply an inter-procedural data-flow analysis using a custom CodeQL configuration. In this analysis, the invocation point acts as a sink, and we trace its value back to a local function or module source ($V_{\text{func\_int}}$, $V_{\text{mod\_int}}$). If a callee is successfully identified within the package by either method, we create the corresponding $E_{\text{int\_call}}$ edge. Any invocation point that cannot be resolved to a local entity through these methods is consequently classified as an external invocation ($V_{\text{invk\_ext}}$). For each of these external invocation points, we analyze the call expression's structure to extract the external dependency's identifier and its API access path. This information is used to create or retrieve a corresponding placeholder node ($V_{\text{mod\_ext}}$ or $V_{\text{func\_ext}}$), and an $E_{\text{ext\_call}}$ edge is established to link the invocation point to it. Crucially, the extracted API access path is stored as metadata on the invocation node itself, forming the foundational input for the interface contract extraction in the next phase.

C. Formal Interface Contract Extraction

Having generated isolated, high-fidelity RSGs in the first phase, the challenge remains to connect them in a way that is both scalable and semantically sound. A simple union of these graphs is insufficient, as it would fail to resolve the symbolic links of inter-package calls. To bridge this gap, we designed a formal abstraction layer: the *interface contract*. The RSG's export anchors (V_{export}) and external invocation vertices ($V_{\text{invk\_ext}}$) provide the explicit structural foundation for extracting the interface contract. The goal of this phase is to extract this contract from each package's RSG, transforming the rich internal details into a concise, formal specification of its external-facing behavior. This contract acts as a "semantic firewall" that decouples a package's internal implementation from its interactions with the wider ecosystem, which is the cornerstone of our pre-compute and combine strategy.

Interface Contract Design. As formally defined in Def. 2, the interface contract $C(P)$ for a package P is a tuple

$\langle \mathcal{M}_E, \mathcal{M}_I \rangle$, which consists of an Export Manifold and an Import Manifest. The contract's foundation is the *API Path*, a structured sequence of operations ($p = \langle op_1, \ldots, op_n \rangle$) drawn from a canonical vocabulary, *Op*. This vocabulary includes primitives like `moduleExport`, `moduleImport(pkg)`, and `getMember(prop)`, designed to create a composable representation of inter-package interactions.

DEF 2: Interface Contract

$C(P) \triangleq \langle \mathcal{M}_E, \mathcal{M}_I \rangle$ where:

- Export Manifold($\mathcal{M}_E$) : $\Pi_{\text{def}} \rightarrow 2^{V_{\text{func\_int}} \cup V_{\text{func\_ext}}}$
- Import Manifest($\mathcal{M}_I$) : $\Pi_{\text{use}} \rightarrow 2^{V_{\text{invk\_ext}}}$
- $\Pi_{\text{def}} \triangleq \{p \in \text{Op}^+ \mid p[0] = \texttt{moduleExport}\}$
- $\Pi_{\text{use}} \triangleq \{p \in \text{Op}^+ \mid p[0] = \texttt{moduleImport(pkg)}\}$
- $\text{Op}^+ \triangleq \{\langle op_1, \ldots, op_n \rangle \mid o_n \in \text{Op}\}$ where:
  - Op $\triangleq$ {`moduleImport(pkg)`, `moduleExport`, `getMember(prop)`, `getReturn()`, `getParameter(idx)`, `getInstance()`}

The Export Manifold ($\mathcal{M}_E$) defines the package's public API surface. It is a mapping from a set of definition paths (Π_{def}) to the set of function entities they expose ($2^{V_{\text{func\_int}} \cup V_{\text{func\_ext}}}$). A definition path is an API path that must begin with the `moduleExport` operation. This design captures diverse JavaScript export patterns. For example in Fig. 2, in `util-lib` package, the standard export of the `process` function and the ref-export of `core.parse` are both represented as distinct paths in Π_{def}, mapping to a node in $V_{\text{func\_int}}$ and $V_{\text{func\_ext}}$ respectively.

The Import Manifest ($\mathcal{M}_I$) is the counterpart that catalogues external dependencies. It is a mapping from a set of use paths (Π_{use}) to the set of external invocation points that use them ($2^{V_{\text{invk\_ext}}}$). A use path is an API path that must begin with a `moduleImport(pkg)` operation, specifying the target dependency. For instance, the call to `utils.parser()` in `app` (Listing 1 of Fig. 2) is formalized into a unique use path in Π_{use}, which is then mapped to its corresponding invocation node in $V_{\text{invk\_ext}}$. This design provides the necessary "plugs" to be connected to the "sockets" defined by the Export Manifolds of other packages.

Interface Contract Implementations. The extraction of the interface contract is implemented as a deterministic process that operates on the rich structure of the pre-computed RSG. This process leverages and abstracts over CodeQL's advanced `ApiGraphs.qll` library to construct the canonical API paths from the vocabulary *Op*.

To build the Export Manifold ($\mathcal{M}_E$), our implementation traverses all export anchor nodes (V_{export}) in the RSG. For each anchor, it analyzes the corresponding source code expression to construct the canonical sequence of operations representing its definition path ($p \in \Pi_{\text{def}}$). This path is then mapped to the target node of the anchor's resolution edge ($E_{\text{std\_exp}}$ or $E_{\text{ref\_exp}}$), formalizing the link between the public API and the underlying function entity.

To build the Import Manifest ($\mathcal{M}_I$), the implementation identifies every external invocation node in $V_{\text{invk\_ext}}$. For each such node, it retrieves the API access path metadata that was stored on it during RSG generation. This metadata is transformed into the canonical API use path sequence ($u \in \Pi_{\text{use}}$). The resulting path is then mapped to the invocation node itself, cataloguing the precise nature of the external call.

This process results in a collection of self-contained RSGs, each annotated with a formal interface contract. The contract provides the unambiguous bridge between a package's internal code and its role in the wider software ecosystem, setting the stage for the final, topology-aware compositional synthesis.

D. Compositional Synthesis of Ecosystem-Scale Call Graph

The final phase synthesizes the individual RSGs into a single coherent Ecosystem-Scale Call Graph (ECG). This is not a simple union of graphs but an intelligent *stitching* process guided by the interface contracts and the package's dependency structure. The complete synthesis algorithm is detailed in Algorithm 1 in Appendix.

The algorithm begins by establishing a processing order that respects the dependency hierarchy. It constructs a package-level dependency graph from project manifest files (e.g., package.json) and performs a *reverse topological sort*. This critical ordering ensures that when any package P_i is processed, the graphs for all of its dependencies have already been fully resolved. For example, in the dependency chain `app` → `util-lib` → `core-lib`, the required processing order is `core-lib`, then `util-lib`, and finally `app`. The algorithm then iterates through this sorted list, maintaining a cache to store the resolved graph for each processed package.

For each package P_i in the sorted list, the algorithm first loads its own isolated RSG. It then sequentially composes this graph with the already resolved graphs of its direct dependencies, which are retrieved from the cache. The core of this process is the *Compose* function, which first performs a structural union of the caller's and callee's graphs. Subsequently, it performs the pivotal interface stitching step. This step resolves symbolic links by matching use paths (Π_{use}) from the caller's Import Manifest ($\mathcal{M}_I$) with definition paths (Π_{def}) from the callee's Export Manifold ($\mathcal{M}_E$).

This path-based matching handles two primary scenarios. For direct path matching, a use path from `app` like $<moduleImport('util-lib'), getMember('process')>$ is directly matched with the corresponding definition path in `util-lib`'s Export Manifold. This creates a new $E_{\text{std\_exp}}$ edge (green dashed arrow in Fig. 2) from the $V_{\text{invk\_ext}}$ node in `app` to the $V_{\text{func\_int}}$ node for `process` in `util-lib`.

For transitive resolution, such as resolving the call to `utils.parser()` in `app`, the algorithm matches the use path to a definition path in `util-lib`'s contract. This contract, however, indicates that the API resolves to an entity imported from `core-lib`. Since the graph for `core-lib` has already been fully resolved, the algorithm transitively creates a direct $E_{\text{ref\_exp}}$ edge from `app`'s invocation node to the `_parse` function's node within `core-lib`'s graph.

This accurately models the true invoke relationship, bypassing the intermediate package. After all dependencies of P_i have been composed, the resulting fully resolved graph is stored in the cache. Once all packages have been processed, the graph associated with the main application P_{app}, is the final ECG.

This three-phase design systematically addresses the challenge of scalable analysis. The RSG generation phase captures detailed information for each package independently. The interface contract extraction then creates formal, abstract boundaries for these packages. Finally, the compositional synthesis algorithm intelligently stitches these pre-computed components together. This process creates a precise, ecosystem-scale CG. The resulting ECG provides a high-fidelity foundation for fine-grained vulnerability analysis at a scale previously impractical. We will empirically evaluate the scalability and precision of this design in the following sections.

IV. Comparative Evaluation

This section comprehensively evaluates VulTracer's accuracy, performance, and advantages over state-of-the-art baselines. We also conduct an ablation study to validate the contributions of our framework's core components. To structure this comparative evaluation, we address the following four research questions (RQs):

RQ1: Accuracy. How accurately does our ECG construction process capture both intra-package and inter-package function calls?

RQ2: Performance. What is the performance and scalability of our three-phase analysis pipeline comparing to traditional whole-program approaches?

RQ3: Effectiveness. How effective is the resulting ECG in reducing FPs in vulnerability analysis compared to existing dependency scanners such as `npm audit`?

RQ4: Ablation. What is the contribution of each core component to VulTracer's effectiveness?

After empirically validating our methodology through the preceding research questions, Section V further presents an ecosystem-scale analysis of vulnerability propagation within the npm registry, thereby demonstrating the distinctive analytical capabilities afforded by ECGs.

A. RQ1: Accuracy of Intra- and Inter-Package Call Resolution

Accurate CG construction is fundamental to function-level vulnerability analysis. Since our approach relies on tracing call paths to determine whether vulnerable functions are reachable, the precision and recall of the underlying CG directly impact the reliability of our analysis. We therefore evaluate VulTracer's intra- and inter-package call resolution accuracy to validate its effectiveness. We evaluate CG accuracy using standard metrics including precision (ratio of correct edges to all reported edges), recall (ratio of correct edges to all actual edges), and F1 score. For inter-package analysis, we measure coverage as the percentage of dynamically observed cross-package calls that are correctly identified.

Due to the absence of complete ground-truth CGs for real-world projects, we adopt a validation methodology based on dynamic analysis. We observe that many open-source npm projects achieve 100% line code coverage through comprehensive test suites. We leverage this by executing these tests and capturing all dynamically invoked call edges as ground-truth datasets. While 100% line coverage does not guarantee exhaustive execution traces, this coverage-guided approach is more realistic and better reflects real-world Node.js usage than synthetic benchmarks. This enables quantitative evaluation of VulTracer against Jelly [13], an enhanced CG generator based on JAM [14].

We identified suitable projects on GitHub by selecting 15 repositories reporting 100% test coverage via CI/CD badges, indicating comprehensive test suites. After locally installing each project and its dependencies, we employed NodeProf [24] on GraalVM [25] to trace function calls during test execution. Due to incompatibilities between GraalVM and certain testing frameworks, we successfully generated dynamic CGs for seven projects, yielding a ground-truth dataset of 6,796 dynamic call edges. The projects span different domains and complexities, including build tools (`gulp`, 33.1K stars), markdown parsers (`markdown-it`, 19.2K stars), and utility libraries (`franc`, 4.2K stars). While test suites achieve 100% coverage of projects' own code, their coverage of third-party dependencies remains minimal. This distinction necessitated a two-level validation strategy. At the intra-package level, dynamically captured CGs served as ground truth for evaluating precision and recall. At the inter-package level, we measure the percentage of dynamically observed cross-package calls correctly identified.

Results. As detailed in Table I, VulTracer significantly outperforms Jelly, achieving an average F_1 score of 0.905 compared to Jelly's 0.731. Notably, VulTracer maintains perfect precision (1.000) across all benchmarks, effectively eliminating the false positives that compromise Jelly's accuracy (0.753 precision). For inter-package analysis, VulTracer identified 27 more valid cross-package calls than Jelly, achieving 65.08% coverage compared to Jelly's 58.67%. Analysis of the results reveals the source of these performance differences. VulTracer's superior recall stems from its robust handling of complex language constructs. Because static CG analysis must infer run-time behavior without actually executing the program, any invocation that depends on run-time construction or indexing (e.g., `obj[someVar]()`) is, in principle, undecidable. For example, in *gulp* all five such dynamically exercised edges are absent from the static results of both tools, reflecting the common upper bound prescribed by Rice's theorem rather than an implementation shortcoming.

VulTracer applies flow- and context-sensitive analysis that accurately separates anonymous callbacks from higher-order invocations, removing the spurious call-graph edges typical of over-approximation. By contrast, Jelly augments its constraint-based analysis with approximate interpretation [26]. These techniques use dynamic instrumentation gathers runtime hints for hard-to-analyze JavaScript features (e.g., `obj[someVar]()`, `eval`). On the *co* benchmark alone, this policy introduces 745 extraneous edges. Although these hints raise recall slightly, the attendant surge in FPs [27] lowers the

TABLE I: Evaluation of intra- and inter-package call resolution accuracy across different tools. VT denotes VulTracer.

Project	Stars	Intra-package						Inter-package	
		Jelly(R)	VT(R)	Jelly(P)	VT(P)	Jelly(F_1)	VT(F_1)	Jelly(Coverage)	VT(Coverage)
gulpjs/gulp	33.1K	0.884	0.884	0.884	**1.000**	0.884	**0.938**	75.84% (113/149)	**83.22%** (124/149)
markdown-it/markdown-it	19.2K	0.484	**0.491**	0.737	**1.000**	0.584	**0.658**	100% (1/1)	100% (1/1)
tj/co	11.9K	**0.993**	0.907	0.168	**1.000**	0.287	**0.951**	37.50% (3/8)	37.50% (3/8)
wooorm/franc	4.2K	0.720	**1.000**	0.947	**1.000**	0.818	**1.000**	4.44% (2/45)	**31.11%** (14/45)
primus/eventemitter3	3.4K	**0.825**	0.819	0.886	**1.000**	0.854	**0.900**	44.58% (42/94)	**45.74%** (43/94)
bcoe/c8	2K	**0.970**	0.921	0.867	**1.000**	0.916	**0.959**	69.35% (86/124)	**71.77%** (89/124)
cosmicanant/recursive-diff	153	0.765	**0.863**	0.780	**1.000**	0.772	**0.926**	-	-
Average		0.806	**0.841**	0.753	**1.000**	0.731	**0.905**	58.67% (247/421)	**65.08%** (274/421)

Bold values indicate the superior result in each comparison pair. **R** denotes Recall, and **P** denotes Precision. Coverage is shown as (covered items/total items) percentage.

average F_1 score by 0.174. By avoiding these FPs, VulTracer maintains a precision of 1.000 across all benchmarks.

Cross-package calls face the same undecidability induced by dynamic properties and higher-order callbacks, yet differences in analysis strategy further amplify the coverage gap. Jelly performs single-package, top-down analysis; once an upstream entry point in the dependency chain is pruned, all downstream edges disappear. In contrast, VulTracer partitions the dependency graph into sub-packages amenable to parallel analysis, performs local reasoning, and then merges the partial graphs into a complete CG, avoiding the single-cut effect. Across the seven projects, VulTracer discovers 27 additional valid cross-package edges, boosting coverage to 65.08%.

By eliminating FPs entirely, VulTracer delivers markedly superior support for JavaScript's complex language features relative to existing approaches. Its few remaining omissions arise from theoretical undecidability, underscoring its balanced advantage in both precision and coverage.

B. RQ2: Performance Evaluation

This evaluation demonstrates the performance and scalability benefits of our composable methodology compared to JAM's monolithic approach (implemented by Jelly). We randomly sampled 100 dependency graphs affected by CVE-2023-32314 (Table XI in Appendix), successfully installing 99 source packages with 26,653 total npm dependencies. The single failure resulted from a dependency resolution error.

JAM employs file-level abstractions to optimize computations within a single npm projects analysis. However, this mechanism has fundamental limitations. Rather than caching final computation results, JAM caches *intermediate artifacts* required for specific data-flow algorithms. These artifacts are reusable only when a file is referenced from multiple call sites within the same analysis run. Because these intermediate artifacts are algorithm-specific and context-dependent, they cannot be transferred across different npm projects or adapted to alternative resolution algorithms. This inability to reuse results across projects causes significant redundancy: in our evaluation, JAM must re-analyze all 26,653 packages involved, even when many identical packages appear across the analyzed projects.

In contrast, VulTracer enables an "analyze-once, reuse-many-times" model by caching final analysis results for unique packages. Across 574 identified propagation paths, VulTracer reduces analysis from 26,653 to just 506 unique packages—a 98% reduction. For example, while stentor-response@1.27.46 involves a dependency graph containing 205 packages, it is only affected by CVE-2023-32314 through a single path involving 4 packages. This allows VulTracer to precisely narrow the analysis scope to these relevant targets, thereby explicitly determining the vulnerability's impact. By comparison, Jelly must analyze all 205 packages from scratch to reach the same conclusion.

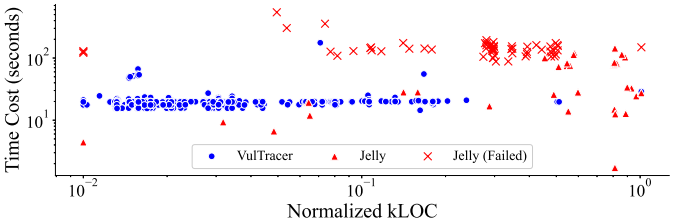

Fig. 3: Time consumption comparison of different tools.

The architectural advantage of VulTracer is evident in its resource efficiency. Under a strict 4 GB memory limit, Jelly achieved a success rate of only 37.37% (37 out of 99 source packages), with the majority failing due to memory exhaustion inherent to monolithic analysis. In contrast, VulTracer demonstrated robust scalability with a 99.41% (503 out of 506 packages) success rate. Although the one-time pre-computation of RSGs required 174 minutes, this cost is mitigated by our reusable design, enabling the final on-demand synthesis to complete in merely 41.87 seconds. This contrasts sharply with Jelly's 31.34 minutes to analyze just the 37 source packages it successfully processed. As Figure 3 illustrates, VulTracer effectively decouples analysis overhead from project complexity (kLOC), maintaining stable performance while Jelly exhibits significant degradation as code size increases.

C. RQ3: Effectiveness in False Positive Reduction

A primary objective of our framework is to enhance the precision of vulnerability auditing by eliminating alerts for

unreachable code. This evaluation quantitatively assesses the effectiveness of VulTracer in reducing FPs compared to package-level dependency scanners. We benchmark our approach against `npm audit`, a widely adopted industry tool also leveraged by services like Dependabot [9]. To ensure a consistent and comparable evaluation, we use the same set of 12 applications from the previous work JAM [14] as our evaluation subjects. Because we cannot create a full vulnerability propagation path from npm audit's output directly, we opt to utilize its API to identify vulnerabilities in the dependency graph we built. By traversing the graph, we obtain all vulnerability propagation paths (P_{vul}). Since npm audit reports each package affected by a vulnerability in the dependency graph as an alarm, we use these alarms as the baseline for our comparison.

Results. As shown in Table II, VulTracer achieves a 94% reduction in FPs, outperforming npm audit, and surpassing JAM's reduction rate of 81% [14]. Specifically, for all target applications, VulTracer identifies 75 vulnerability propagation paths and 53 vulnerability alarms (49 unique ones), which means that only 9% of the packages (49 out of 532) in the dependency graphs are potentially affected, significantly reducing the triage overhead.

Moreover, we manually validate each alarm and find that, among a total of 75 different vulnerability propagation paths, only 11 truly affect the source package, and only 21 of the 53 alarms are true positives (TPs). Due to npm's semantic versioning ranges, dependency versions are resolved dynamically at installation time rather than being statically fixed (e.g., `^1.2.0` will match `1.2.5` today but may match `1.3.0` after a new release). This makes it infeasible to perfectly reproduce JAM's prior experimental environment. Therefore, we evaluate our approach by analyzing the same applications and comparing the final results. After scrutinizing the 75 paths using VulTracer, the results show that VulTracer correctly identifies all vulnerable paths and reduces FPs from 32 to 2. The two remaining FPs are caused by the dynamic feature of JavaScript (e.g., high-order function and dynamic function call), which is a common challenge for all static analysis.

TABLE II: Comparison of vulnerability audit results of VulTracer and npm audit.

Target	# Pkgs	# P_{vul}	# Alarm	npm audit		VulTracer	
				TP	FP	TP	FP
makeappicon@1.2.2	14	6	2	2	0	2	0
toucht@0.0.1	25	5	4	0	4	0	0
spotify-terminal@0.1.2	85	16	6	3	3	3	0
ragan-module@1.3.0	56	1	3	0	3	0	0
npm-git-snapshot@0.1.1	36	2	4	0	4	0	0
nodetree@0.0.3	5	6	2	0	2	0	0
jwtnoneify@1.0.1	79	7	4	0	4	0	0
foxx-framework@0.3.6	61	1	3	0	3	0	0
npmgenerate@0.0.1	23	5	4	4	0	4	0
smrti@1.0.3	59	1	3	0	3	0	0
writex@1.0.4	46	16	8	6	2	6	2
openbadges-issuer@0.4.0	43	9	10	6	4	6	0
Total	532	75	53 (49)	21	32	21	2

Pkgs refers to the count of packages within the dependency graph. P_{vul} indicates the number of vulnerability propagation path. **Alarm** denotes all vulnerable packages in the dependency graph.

D. RQ4: Ablation

To evaluate our core design choices, we conducted an ablation study by disabling three key components:

- **VulTracer (Full Setup):** As detailed in Section III.
- **VT-NoContract:** Removes formal interface contracts (Section III-C), using direct function name matching for graph merging instead.
- **VT-SimpleAPI:** Restricts the API vocabulary (Def. 2) by excluding primitives for complex data flows: getReturn(), getParameter() and getInstance().
- **VT-NoRTS:** Disables Reverse Topological Sort (Algorithm 1), processing packages in naive dependency order.

TABLE III: Ablation study results for VulTracer.

Variant	Intra F_1	Inter Cov.	Resolved
VULTRACER (Full)	**0.905**	**65.08%**	**274**
VT-NoContract	0.905	19.95%	84
VT-SimpleAPI	0.905	48.22%	203
VT-NoRTS	0.905	58.19%	245

Note: **Intra F_1**: Avg. Intra-Package F_1 score; **Inter Cov.**: Inter-Package Coverage; **Resolved**: Number of calls resolved (out of 421 total).

Table III shows results on the ground-truth dataset from RQ1. The full VulTracer achieves 65.08% inter-package coverage, substantially outperforming all ablated variants (19.95%, 48.22%, and 58.19%), while maintaining stable intra-package accuracy ($F_1 = 0.905$). This confirms our design specifically enhances cross-package resolution.

The 45.13% drop in VT-NoContract demonstrates that formal Interface Contracts are essential for semantic matching beyond naive name-based approaches. VT-SimpleAPI's 16.86% reduction shows that expressive API primitives are critical for modeling complex data flows. The 6.89% degradation in VT-NoRTS validates the importance of correct processing order for resolving intricate re-export patterns.

V. Ecosystem-Scale Vulnerability Propagation Evaluation

In this section, we leverage VulTracer to conduct a large-scale analysis of the npm ecosystem. The goal of this section is to provide broad and empirical insights into the nature of software supply chain risk. To guide this investigation, we focus on two fundamental research questions:

RQ5. How often do package-level alerts over-approximate real reachability?

RQ6. What factors affect the attenuation along dependency hops?

Our investigation examines vulnerability propagation at two distinct perspectives. First, an analysis at the *single-hop* level is conducted on direct dependencies to uncover the root causes of over-approximation and the primary factors of attenuation. The analysis then proceeds to the *multi-hop* level, where we investigate how these initial effects compound across transitive dependency chains, thereby providing a comprehensive, ecosystem-scale perspective on both research questions.

A. Experimental Setup

Our experiments were conducted on a machine with an Intel Xeon Silver 4210 CPU and 187 GB RAM. Based on preliminary analysis, we imposed a 120-second time and 4 GB memory limit for each package analysis task. To enable efficient large-scale vulnerability impact assessment, we pre-processed data for each step in the VulTracer workflow separately. For acceleration, we utilized Apache Spark with a Yarn queue comprising 3,000 cores and 6,144 GB RAM, maintaining consistent environment configurations throughout all evaluations.

Datasets. To evaluate our approach at the ecosystem scale, we constructed two datasets. The first, npm package and dependency dataset (DS_{npm}) is constructed by sourcing the complete dependency graph from Google's Open Source Insights [28] and we follow the RScouter [29] to collect metadata and source code for all packages published before December 31, 2024. This process yields a dataset that encompasses 3,267,273 unique packages, 34,685,976 distinct versions, and over 900 million documented dependency links. The second dataset (DS_{CVE}) was curated using a two-dimensional strategy to ensure both ecosystem impact and vulnerability diversity. First, we selected vulnerabilities from the top 10 most downloaded npm packages, yielding six high-impact CVEs. Second, we aligned with the 2024 CWE-Top-25 [30] (a ranking of the most dangerous software weaknesses), prioritizing the most severe and recent CVEs for each applicable category, yielding 21 distinct CVEs. After manual verification, DS_{CVE} comprises 27 unique CVEs with precisely identified vulnerable functions (Vul_{func}). The full selection criteria and CVE lists are detailed in Appendix A (Tables VII and VIII).

Terminology and Hop-Set Partitioning. Throughout this section, we use the symbols d_0 and d_k to distinguish between *directly vulnerable* packages and their *k-hop dependents* at various distances in the dependency graph $\mathcal{G}$:

- d_0 (directly vulnerable package): A package version containing at least one function identified as vulnerable by a CVE.
- d_1 (one-hop dependent): A package version whose `package.json` declares *at least one direct dependency* on a d_0 package version. In the npm dependency graph, d_1 nodes are therefore exactly one edge away from the corresponding d_0 node.
- d_k (k-hop dependent): A package p where the shortest dependency path to a vulnerable package v is of length k. We define $d_k = \{\, p \in V \mid \text{dist}_{\mathcal{G}}(p, v) = k \,\}$ for $k \geq 1$.

Our analysis is divided into two stages: a *single-hop analysis* ($d_1 \rightarrow d_0$) to precisely attribute propagation to direct dependencies, followed by a *multi-hop study* ($k \geq 2$) to observe transitive propagation decay.

B. Single-Hop Analysis ($d_1 \rightarrow d_0$)

We first restrict our attention to the $d_1 \rightarrow d_0$ edges, allowing for precise attribution of vulnerability propagation to a single dependency relation. This eliminates confounding factors introduced by longer dependency chains.

To formalize our analysis, we model a package p as a collection of modules, denoted by the set $M(p)$. Each module $m \in M(p)$ in turn contains a set of functions, $F(m)$. The set of known vulnerable functions within the source package d_0 is represented by Vul_{func}. The interaction between the dependent and the dependency is captured by two key relational sets: $M_{\text{imp}}(d_1, d_0)$, the set of modules that d_1 imports from d_0, and for each imported module m, $F_{\text{call}}(d_1, m)$, the set of functions that d_1 calls from it. The set of all functions called by d_1 from modules imported from d_0 is defined as $F_{\text{called}}(d_1, d_0)$:

$$F_{\text{called}}(d_1, d_0) \triangleq \bigcup_{m \in M_{\text{imp}}(d_1, d_0)} F_{\text{call}}(d_1, m) \tag{1}$$

With this notation, the three conditions for propagation can be expressed concisely:

$$C_{\text{mod}} \iff M_{\text{imp}}(d_1, d_0) \neq \emptyset \tag{2}$$

$$C_{\text{func}} \iff F_{\text{called}}(d_1, d_0) \neq \emptyset \tag{3}$$

$$C_{\text{vuln\_func}} \iff F_{\text{called}}(d_1, d_0) \cap \text{Vul}_{\text{func}} \neq \emptyset \tag{4}$$

Thus, the overall propagation predicate is the logical conjunction of these three conditions:

$$P(d_0, d_1) \iff C_{\text{mod}} \wedge C_{\text{func}} \wedge C_{\text{vuln\_func}} \tag{5}$$

Our analysis of 27 CVEs covers 1,679 directly vulnerable versions (d_0) across 25 distinct packages. These versions are listed as direct dependencies by 703,896 one-hop (d_1) package versions. The results (fully detailed in Table IX in Appendix, with high-impact examples in Table IV) show a significant reduction at each stage of our analysis.

Answer to RQ5: Over-approximation of Package-Level Alerts. Our single-hop analysis provides a direct answer to RQ5, revealing that package-level alerts grossly over-approximate true vulnerability reachability. At the single-hop level, package-level alerts were found to over-approximate reachability by an average of 53.66% in the high-impact CVEs (Table IV) and 67.51% globally (Appendix Table IX). These two tables provide a detailed breakdown of attenuation at each analysis stage. Specifically, only 46.34% of one-hop dependents in the high-impact dataset and 32.49% globally are confirmed to call a known vulnerable function (satisfying $C_{\text{vuln\_func}}$). The primary reasons for this over-approximation are unused dependencies (failure to meet C_{mod}) and the use of only non-vulnerable functions from the dependency (failure to meet $C_{\text{vuln\_func}}$ despite meeting C_{func}). Regarding unused dependencies, 22.80% of d_1 packages in the high-impact set and 42.28% globally fail the C_{mod} condition, meaning they declare a vulnerable dependency but never import any module from it. For CVE-2021-23337 in `lodash`, over a third of the dependent packages ($\#d_1$ - $\#C_{\text{mod}}$, 131,933) fall into this category. This underscores the need for function-level analysis to reduce alert fatigue and prioritize real threats.

Answer to RQ6: Factors of Single-Hop Attenuation. Our analysis reveals that the degree of this sharp attenuation is not uniform across different vulnerable packages. Instead, it is primarily determined by two factors: the scope of the vulnerable

TABLE IV: Single-hop reachability analysis and attenuation for High-impact vulnerabilities. The full detailed results for all 27 CVEs are provided in Table IX in the Appendix.

CVE ID	Package Name	# Vul_{func}	# d_0	# d_1	# C_{mod}	# C_{func}	# $C_{\text{vuln\_func}}$
CVE-2021-23337	lodash	1	100	396,112	264,179 (66.69%)	244,130 (61.63%)	11,574 (2.92%)
CVE-2022-3517	minimatch	7	26	38,112	28,667 (75.22%)	15,791 (41.43%)	15,791 (41.43%)
CVE-2016-10540	minimatch	5	23	10,341	9,211 (89.07%)	3,528 (34.12%)	3,528 (34.12%)
CVE-2022-25883	semver	14	74	139,257	111,138 (79.81%)	102,209 (73.40%)	73,314 (52.65%)
CVE-2017-16137	debug	1	55	70,297	54,098 (76.96%)	51,425 (73.15%)	50,454 (71.77%)
CVE-2017-20165	debug	1	42	39,365	29,702 (75.45%)	29,583 (75.15%)	29,576 (75.13%)
Average	-	-	-	-	77.20%	59.81%	46.34%

library's API and the usage frequency of the vulnerable function. Notably, libraries with broad-APIs demonstrate significantly different patterns compared to those with narrow APIs.

Broad-API Libraries (lodash): CVE-2021-23337 in `lodash` represents an extreme case of attenuation. As a comprehensive utility library with hundreds of functions, developers often use only a small fraction of its capabilities. While a majority of dependents (66.69%) import the package, a minuscule 2.92% of the d_1 packages actually invoke the vulnerable `template` function. This exemplifies how a vulnerability in a specialized function within a large library has a much smaller "blast radius" than package-level scanning would report.

Narrow-API Libraries (debug): In contrast, vulnerabilities in the `debug` package exhibit a much higher propagation rate. `debug` has a focused API centered on its main debugging functionality. Consequently, a large majority of developers who import the library are very likely to use its core, vulnerable function. For both CVE-2017-16137 and CVE-2017-20165, over 98% of the dependents that import from `debug` also call the vulnerable function, leading to a final propagation rate of over 70% among all direct dependents.

The `semver` and `minimatch` packages represent intermediate cases. Their APIs are more extensive than `debug`'s but more focused than `lodash`'s. For `semver` (CVE-2022-25883), a high percentage of packages that import it also use its functions, and a substantial number of those call one of the 14 vulnerable functions, resulting in a high final propagation rate of 52.65%. For the `minimatch` CVEs, there is a notable drop-off between importing a module and calling a function. The dependents that do call a function from `minimatch`, the probability of it being vulnerable is 100% in these cases, as the vulnerabilities reside in the package's core pattern-matching logic. We observed identical attenuation trends in the diversity CVEs (detailed in Appendix Table IX), confirming these findings are consistent across different vulnerability types.

To further analyze the usage patterns driving attenuation in broad-API libraries. We performed a detailed analysis of `lodash` function calls, summarized in Table V. Across 244,130 package versions, we observed 7,830,664 calls to 242 distinct `lodash` functions. The call distribution reveals that a small number of functions account for a large proportion of usage. The most frequently called functions, `forEach` (10.66%), `isFunction` (9.99%), and `get` (9.62%), account for over 30% of all calls. In contrast, the vulnerable function `template` is ranked 49th, constituting only 0.30% of total calls. This long-tail distribution demonstrates that library usage is often shallow despite its wide adoption. Consequently, the impact of vulnerabilities in less common functions is limited.

TABLE V: Frequency and dependency analysis of `lodash` functions. N_{call} represents the total call count. D_{total} is the number of downstream packages including all versions, while D_{uniq} is the count of unique downstream package names.

No.	F_{name}	# N_{call} (%)	Downstream Dependencies (d_1)	
			# D_{total} (%)	# D_{uniq} (%)
1	forEach	834,950 (10.66%)	48,224 (19.75%)	4,363 (20.00%)
2	isFunction	782,490 (9.99%)	41,857 (17.15%)	3,741 (17.15%)
3	get	753,685 (9.62%)	37,695 (15.44%)	2,066 (9.47%)
4	map	457,311 (5.84%)	60,801 (24.91%)	5,513 (25.27%)
5	isEmpty	423,905 (5.41%)	48,974 (20.06%)	3,084 (14.14%)
6	isObject	335,162 (4.28%)	44,844 (18.37%)	3,650 (16.73%)
7	isString	326,310 (4.17%)	59,572 (24.40%)	4,542 (20.82%)
8	cloneDeep	283,015 (3.61%)	44,765 (18.34%)	2,829 (12.97%)
9	isUndefined	271,190 (3.46%)	27,684 (11.34%)	2,551 (11.69%)
10	filter	213,382 (2.73%)	36,464 (14.94%)	2,919 (13.38%)
48	contains	23,728 (0.30%)	7,413 (3.04%)	801 (3.67%)
49	**template**	23,148 **(0.30%)**	11,574 **(4.74%)**	1,150 **(5.27%)**
50	isNaN	22,903 (0.29%)	8,475 (3.47%)	437 (2.00%)

Furthermore, co-occurrence analysis shows that `template` is most frequently used with functions like `get` (95.27%), `isString` (72.82%), and `map` (70.44%). This pattern indicates that `template` primarily serves string generation tasks where data is dynamically extracted from objects. These findings reveal how the vulnerable function is actually used in practice. They also explain why the `template` function's impact remains limited despite `lodash`'s widespread adoption.

C. Multi-Hop Propagation Study ($k \geq 2$)

Building on the single-hop analysis, we extend our evaluation to trace vulnerability propagation through longer dependency chains ($k \geq 2$). This stage aims to quantify the decay in reachability over multiple hops and to contrast the impact scope as measured by function-level analysis versus traditional package-level dependency graph traversal. Our analysis starts from the set of d_1 packages confirmed to be affected at the function level. These packages serve as the starting point for tracing further transitive propagation. Finally, from these packages, we identified 9,868,514 potential propagation paths (deduplicated) involving 1,663,634 package versions (also deduplicated) by traversing the npm dependency graph. A

TABLE VI: Comparative analysis of transitive vulnerability propagation: package-Level vs. function-Level (High-impact). The full detailed results for all 27 CVEs are provided in Table X in the Appendix.

CVE ID	Paths	Affected Library			Affected Version			Avg. Hop Distance		Max Hop Distance	
		# P-L	# F-L	RP	# P-L	# F-L	RP	# P-L	# F-L	# P-L	# F-L
CVE-2021-23337	285,622	17,267	4,826	27.95%	166,554	37,220	22.35%	3.34	0.61	13	6
CVE-2022-3517	497,595	21,775	1,211	5.56%	286,731	22,557	7.87%	4.15	0.12	13	5
CVE-2016-10540	127,976	14,499	649	4.48%	84,886	7,916	9.33%	3.65	1.14	12	5
CVE-2022-25883	2,096,181	48,192	3,863	8.02%	595,078	74,823	12.57%	7.48	1.09	32	7
CVE-2017-16137	6,663,049	50,444	23,107	45.81%	711,199	286,679	40.31%	7.39	2.05	16	8
CVE-2017-20165	6,184,586	39,460	18,729	47.46%	488,064	208,227	42.66%	7.62	2.08	15	8

P-L indicates package-level analysis, **F-L** indicates function-level analysis, and **RP** represents the relative proportion of **F-L** to **P-L**.

propagation path is defined as a sequence of packages (d_0, d_1, d_2,$\cdots$,d_k) where each d_{i+1} depends on d_i. To analyze these paths efficiently, we pre-computed RSGs and interface contracts for all package versions. Then, when analyzing any specific propagation path, we composed its ECG by combining these pre-computed results based on the dependency relationships between packages in that path.

We employ an iterative, breadth-first search algorithm to trace reachability. A package d_{k+1} is marked as transitively affected if and only if it depends on an already affected package d_k and meets the three propagation conditions (C_{mod}, C_{func}, and a transitive version of $C_{\text{vuln\_func}}$) for that dependency link. This expansion repeats until no new packages can be marked as affected, yielding the complete set of transitively impacted packages. To manage the complexity of ecosystem-scale analysis and avoid issues like path explosion from cyclic or diamond dependencies, our analysis focuses on establishing reachability via the shortest possible call chain between any two packages in the dependency path.

Answer to RQ5: Amplified Over-approximation in the Ecosystem. While the single-hop analysis revealed that over half of direct-dependency alerts are FPs, the multi-hop results show this effect is amplified across the ecosystem. At the package level, the 27 vulnerabilities transitively affect a vast number of packages, with CVE-2017-16137 and CVE-2017-20165 each having over six million potential propagation paths. However, function-level analysis demonstrates that the vast majority of these paths do not represent a real threat. On average, the number of affected libraries identified by function-level analysis is only 31.72% and 34.74% for package and version count respectively. This means that function-level analysis filters out 68.28% of transitively affected packages flagged by package-level tools. For instance, in the case of CVE-2022-3517 (`minimatch`), our analysis dramatically reduces the number of affected libraries from 21,775 to 1,211, a reduction of 94.44%. Similarly, for CVE-2021-23337 (`lodash`), the number of affected libraries drops by 72.05%. This confirms that the initial imprecision at the first hop creates a rapidly expanding cascade of false-positive propagation paths.

Answer to RQ6: Transitive Attenuation and Propagation Decay. As the final number of affected packages and the propagation path lengths presented in Table VI (and fully detailed in Appendix Table X) shows, the attenuation factors identified at the single-hop level amplified at each transitive step, causing vulnerability propagation to decay rapidly. Specifically, at the package level, vulnerabilities appear to travel deep into the dependency graph, with average path lengths ranging from 3.34 to 7.62 (for the high-impact CVEs) and reaching a maximum of 32 hops for CVE-2022-25883. In contrast, function-level analysis reveals that actual propagation is much shallower. The average function-level hop distance for a reachable vulnerability is notably short, ranging from 0.12 to 2.08 in the high-impact set, and the maximum observed propagation was only 8 hops. This significant difference in path depth indicates that even when a vulnerability does propagate transitively, its actual reach is typically limited to a few steps from the source.

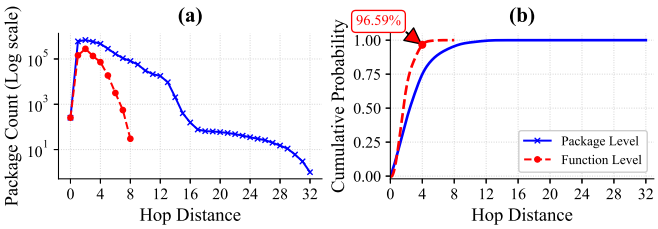

Fig. 4: Package- vs. Function-level propagation decay: (a) Count per hop; (b) CDF.

Fig. 4 provides a direct visualization of the decay dynamics in two complementary views. Panel (a) plots the number of newly affected packages per hop on a log scale, illustrating the magnitude of attenuation at each hop. The significant vertical gap between the package-level (blue line) and function-level (red line) curves reveals the over-approximation at every hop. Moreover, the function-level propagation, shows a clear turning point, as the number of newly affected packages starts to decrease sharply from the third hop. Similarly, as Panel (b) shows, the function-level curve rises sharply and rapidly approaches its maximum and then plateaus, demonstrating that the scope of true impact is established very early (96.59% of all transitively affected packages are identified within just four hops ($k \leq 4$)). In contrast, the package-level impact curve rises much more slowly, requiring many more hops to account for the full set of affected packages.

Taken together, these results demonstrate that the API surface and specific function usage patterns are the dominant

factors limiting a vulnerability's "blast radius". This attenuation effect is magnified at each step in a dependency chain, causing a rapid decay in true vulnerability propagation.

VI. Discussion

A. Practical Implications

Our findings offer actionable guidance for enhancing software supply chain security. By demonstrating the feasibility of a precise, function-level reachability analysis at scale, this work provides a clear path forward for developers, security teams, and tool vendors.

For Developers and Security Teams. The primary implication is the need for *reachability-driven vulnerability triage*. Alerts from SCA tools should be prioritized not by presence, but by evidence of reachability. A reachable vulnerability in a direct dependency warrants immediate remediation, whereas alerts for unreachable functions in transitive dependencies represent minimal risk, allowing for the strategic allocation of engineering resources. Our analysis of "dead" dependencies (C_{mod} failures in Table IV) further highlights the importance of dependency hygiene as a direct method for reducing both attack surface and alert fatigue.

For SCA Tool Vendors. Our results present a clear mandate to move beyond package-level scanning. Integrating static function-level call-graph analysis is critical to reducing alert fatigue and increasing the value of SCA tools. By distinguishing between a specious vulnerability (dependency exists) and a reachable vulnerability (vulnerable function is called), tools can provide actionable intelligence rather than overwhelming reports. This would transform SCA from a system that often frustrates developers with low-signal warnings into a valuable tool to precisely identify and prioritize genuine security risks.

B. Limitations

Call Graph Construction and Evaluation. Our analysis is based on a purely static approach. However, modern JavaScript applications frequently employ dynamic features that are intractable for static analysis to resolve perfectly. For instance, function calls using computed property access, such as `Obj[var]()`, pose a significant challenge. Although our data-flow analysis is designed to resolve some of these dynamic scenarios, capturing all possible runtime behaviors statically is an intractable problem. Consequently, our function-level results on vulnerability propagation should be interpreted as a conservative lower bound. TPs that rely on highly dynamic behavior may be missed.

This challenge also extends to evaluation. At present, the JavaScript/TypeScript ecosystem lacks a comprehensive, project-level "gold standard" benchmark for validating CGs at scale. We therefore established a pragmatic ground truth by dynamically tracing test suites with 100% coverage. While this provides a high-quality baseline, it does not guarantee complete *path coverage*, as function calls nested within complex conditional logic may remain unexercised. However, given that these comprehensive tests cover nearly all statements and branches, such unobserved calls are expected to be rare, making our dynamic traces a robust, albeit imperfect, foundation for evaluation.

Scalability and Analysis Timeouts. To enable large-scale analysis, our methodology imposes strict time and memory limits on the processing of each package. While this design maximizes scalability, a small percentage of packages may fail to complete the analysis within these constraints. Particularly those packages with extreme complexity or obfuscated code (e.g., `@theia/core`). This may lead to an underestimation of vulnerability propagation along paths that involve these complex, timed-out packages. However, these failures represent a minimal fraction of the millions of packages in our dataset, and their impact on the overall statistical findings is considered negligible.

VII. Related Work

Vulnerability Propagation Analysis. Research on vulnerability propagation has evolved from package-level assessments to function-level analysis. Package-level studies [7], [31]–[34] revealed widespread vulnerability prevalence [35]–[37] and demonstrated extensive propagation through transitive dependencies [10], [31], [37]. Complementary work examined intra-project vulnerability distribution [38]. However, package-level analysis overestimates risk by neglecting reachability [39], motivating finer-grained approaches [17], [40]–[42]. Empirical evidence confirms this: over half of indirect dependencies in Crates.io may be unused [40], while only 25.7% of vulnerable functions are reachable in Maven [41]. Ruan et al. [42] introduced whole-ecosystem call-graph analysis for Maven to quantify propagation impact. Within the npm ecosystem, function-level vulnerability reachability analysis has been limited in scale. Early work proposed fine-grained dependency networks using call graphs [43] or manual analysis [39], confirming that most projects do not invoke vulnerable functions even when depending on vulnerable packages. Recent tools face scalability or accuracy challenges: JAM [14] works at small scale but struggles to scale to ecosystem-level security scanning, while SōjiTantei [44] analyzed only 780 clients and faces accuracy limitations with modern JavaScript (e.g., ES6). Other work, such as VulEval [45], focuses on benchmarking detection tools rather than ecosystem-wide propagation. In contrast, VulTracer enables the first ecosystem-scale, function-level empirical study of real-world vulnerability impact in npm.

Call Graph Construction. Function-level analysis requires CG construction, which is challenging for JavaScript. Dynamic analysis [24], [40], [46], [47] is precise but lacks coverage for ecosystem studies [48]. Static analysis is more common, despite inherent trade-offs between performance and precision. Lightweight methods [49]–[52] are fast but may be incomplete, whereas more precise techniques incorporating taint analysis are often too expensive to scale [18], [53]. To balance scalability and precision, "divide-and-conquer" strategies have been proposed. Frankenstein [15] stitches pre-computed call graphs for Java using Class Hierarchy Analysis, but this approach is incompatible with JavaScript's dynamic

features such as prototype mutation. For JavaScript, JAM [14] and its reference implementation Jelly [13] employ file-level caching of intermediate data-flow artifacts. However, these artifacts are algorithm-specific and context-dependent, limiting reuse to within a single project analysis. This forces JAM to re-analyze shared dependencies across different projects, hindering ecosystem-scale scalability. VulTracer addresses this by caching package-level RSGs as final, self-contained analysis results that enable cross-project reuse and efficient dependency chain analysis. This design achieves a balance between scalability and precision for large-scale npm ecosystem analysis.

VIII. Conclusions

In this work, we design a novel vulnerability propagation analysis framework VulTracer, which focuses on identifying the true impact scope of known vulnerabilities at the function level. The experimental results show it has better performance and scalability than state-of-the-art tools. Moreover, we conduct an ecosystem-scale empirical study, covering around 3 million packages, to comprehensively understand the true threat of known vulnerabilities in npm ecosystem. Our analysis reveals several significant findings, which can not only benefit developers in understanding the true security threats of vulnerabilities but also guide future research in this field.

IX. Ethics Considerations

Our work does not raise any ethical issues. During the data collection process from the official npm registry, we utilize the CouchDB replication service [54] as our API interface [55] to avoid extra load on the normal service. Moreover, we strictly limit the frequency of API requests and speed of package downloads. These precautions ensure efficient data collection while avoiding over-stressing the server, allowing us to download tens of millions of packages in a month.

Acknowledgment

This research was supported in part by the National Key Research and Development Program of China (Grant No. 2024YFE0203800), the Young Scientists Fund of the National Natural Science Foundation of China (Grant No. 62402277), and the National Natural Science Foundation of China (Grant No. 62572266).

References

[1] npm, https://www.npmjs.com.
[2] Sonatype, "State of the Software Supply Chain Report," https://www.sonatype.com/state-of-the-software-supply-chain, 2024.
[3] A. Decan, T. Mens, and M. Claes, "On the topology of package dependency networks: A comparison of three programming language ecosystems," in *Proccedings of the 10th European Conference on Software Architecture (ECSA)*, 2016, pp. 1–4.
[4] X. Chen, R. Abdalkareem, S. Mujahid, E. Shihab, and X. Xia, "Helping or not helping? why and how trivial packages impact the npm ecosystem," *Empirical Software Engineering*, vol. 26, pp. 1–24, 2021.
[5] R. Abdalkareem, V. Oda, S. Mujahid, and E. Shihab, "On the impact of using trivial packages: An empirical case study on npm and pypi," *Empirical Software Engineering*, vol. 25, pp. 1168–1204, 2020.
[6] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*, 2017, pp. 385–395.
[7] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. IEEE, 2022, pp. 672–684.
[8] Proxies are complicated: RCE vulnerability in a 3 million downloads/week NPM package, https://httptoolkit.com/blog/npm-pac-proxy-agent-vulnerability.
[9] Dependabot, https://github.com/dependabot.
[10] J. Cox, E. Bouwers, M. Van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, vol. 2. IEEE, 2015, pp. 109–118.
[11] W. Enck and L. Williams, "Top five challenges in software supply chain security: Observations from 30 industry and government organizations," *IEEE Security & Privacy*, vol. 20, no. 2, pp. 96–100, 2022.
[12] Y. Wang, P. Sun, L. Pei, Y. Yu, C. Xu, S.-C. Cheung, H. Yu, and Z. Zhu, "Plumber: Boosting the propagation of vulnerability fixes in the npm ecosystem," *IEEE Transactions on Software Engineering*, vol. 49, no. 5, pp. 3155–3181, 2023.
[13] Jelly, https://github.com/cs-au-dk/jelly.
[14] B. B. Nielsen, M. T. Torp, and A. Møller, "Modular call graph construction for security scanning of node. js applications," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021, pp. 29–41.
[15] M. Keshani, G. Gousios, and S. Proksch, "Frankenstein: fast and lightweight call graph generation for software builds," *Empirical Software Engineering*, vol. 29, no. 1, p. 1, 2024.
[16] npm SemVer Calculator, https://www.npmjs.com/package/semver.
[17] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the attack surface of node. js applications," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020, pp. 121–134.
[18] B. B. Nielsen, B. Hassanshahi, and F. Gauthier, "Nodest: feedback-driven static analysis of node. js applications," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 455–465.
[19] Babel, https://github.com/babel/babel.
[20] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, "Correlation tracking for points-to analysis of javascript," in *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2012, pp. 435–458.
[21] S. Wei and B. G. Ryder, "Adaptive context-sensitive analysis for javascript," in *29th European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015, pp. 712–734.
[22] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 423–434, 2013.
[23] E. Lin, I. Koishybayev, T. Dunlap, W. Enck, and A. Kapravelos, "Untrustide: Exploiting weaknesses in vs code extensions," in *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*. Internet Society, 2024.
[24] nodeprof.js, https://github.com/Haiyang-Sun/nodeprof.js.
[25] GraalVM, https://github.com/oracle/graaljs.
[26] Jelly Approximate Interpretation, https://github.com/cs-au-dk/jelly/blob/6dd7a91c/src/approx/approx.ts.
[27] Issue of Jelly, https://github.com/cs-au-dk/jelly/issues/18#issuecomment-2082106829.
[28] Google's Open Source Insights, https://deps.dev.
[29] Y. Gu, L. Ying, Y. Pu, X. Hu, H. Chai, R. Wang, X. Gao, and H. Duan, "Investigating package related security threats in software registries," in *2023 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2023, pp. 1578–1595.
[30] The MITRE Corporation, "2024 CWE Top 25 Most Dangerous Software Weaknesses," https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25, Common Weakness Enumeration, 2024, accessed: 2025-11-19.
[31] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 181–191.

[32] G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma, and D. Lo, "Out of sight, out of mind? how vulnerable dependencies affect open-source projects," *Empirical Software Engineering*, vol. 26, pp. 1–34, 2021.

[33] A. Zerouali, T. Mens, A. Decan, and C. De Roover, "On the impact of security vulnerabilities in the npm and rubygems dependency networks," *Empirical Software Engineering*, vol. 27, no. 5, p. 107, 2022.

[34] M. Kluban, M. Mannan, and A. Youssef, "On detecting and measuring exploitable javascript functions in real-world applications," *ACM Transactions on Privacy and Security*, vol. 27, no. 1, 2024.

[35] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," in *Proceedings 2017 Network and Distributed System Security Symposium (NDSS)*, 2017.

[36] A. Gkortzis, D. Feitosa, and D. Spinellis, "Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities," *Journal of Systems and Software*, vol. 172, p. 110653, 2021.

[37] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium (USENIX Security)*, 2019, pp. 995–1010.

[38] B. Liu, G. Meng, W. Zou, Q. Gong, F. Li, M. Lin, D. Sun, W. Huo, and C. Zhang, "A large-scale empirical study on vulnerability distribution within projects and the lessons learned," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1547–1559.

[39] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 559–563.

[40] J. Hejderup, M. Beller, K. Triantafyllou, and G. Gousios, "Präzi: from package-based to call-based dependency networks," *Empirical Software Engineering*, vol. 27, no. 5, p. 102, 2022.

[41] Y. Wu, Z. Yu, M. Wen, Q. Li, D. Zou, and H. Jin, "Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1046–1058.

[42] B. Ruan, Z. Lin, J. Liu, C. Zhang, K. Ji, and Z. Liang, "An accurate and efficient vulnerability propagation analysis framework," *arXiv preprint arXiv:2506.01342*, 2025.

[43] J. Hejderup, A. van Deursen, and G. Gousios, "Software ecosystem call graph for dependency management," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2018, pp. 101–104.

[44] B. Chinthanet, R. G. Kula, R. E. Zapata, T. Ishio, K. Matsumoto, and A. Ihara, "Sōjitantei: Function-call reachability detection of vulnerable code for npm packages," *IEICE TRANSACTIONS on Information and Systems*, vol. 105, no. 1, pp. 19–20, 2022.

[45] X.-C. Wen, X. Wang, Y. Chen, R. Hu, D. Lo, and C. Gao, "Vuleval: Towards repository-level evaluation of software vulnerability detection," *arXiv preprint arXiv:2404.15596*, 2024.

[46] Z. Herczeg, G. Lóki, and Á. Kiss, "Towards the efficient use of dynamic call graph generators of node. js applications," in *14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. Springer, 2020, pp. 286–302.

[47] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 488–498.

[48] C.-A. Staicu, M. Pradel, and B. Livshits, "Synode: Understanding and automatically preventing injection attacks on node. js." in *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, vol. 1, 2018, pp. 12–52.

[49] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript ide services," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 752–761.

[50] M. Madsen, B. Livshits, and M. Fanning, "Practical static analysis of javascript applications in the presence of frameworks and libraries," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE)*, 2013, pp. 499–509.

[51] M. Madsen, F. Tip, and O. Lhoták, "Static analysis of event-driven node. js javascript applications," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 505–519, 2015.

[52] S. Li, M. Kang, J. Hou, and Y. Cao, "Mining node. js vulnerabilities via object dependence graph and query," in *31st USENIX Security Symposium (USENIX Security)*, 2022, pp. 143–160.

[53] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel, "Extracting taint specifications for javascript libraries," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 198–209.

[54] CouchDB Replication, https://docs.couchdb.org/en/stable/replication/intro.html.

[55] npm Replication API, https://github.com/npm/registry/blob/master/docs/REPLICATE-API.md.

[56] Google OSV, https://osv.dev/.

APPENDIX

A. CVE Selection Criteria and Workflow

To construct a robust evaluation dataset (DS_{CVE}), we employed a two-dimensional selection strategy designed to balance ecosystem-wide impact with vulnerability diversity. First, we targeted the top 10 most downloaded npm packages in 2024, selecting six high-impact CVEs affecting four widely used libraries: `lodash`, `debug`, `semver`, and `minimatch` (detailed in Table VII). These packages exhibit massive transitive usage, providing an ideal setting for measuring the potential "blast radius" of vulnerabilities.

Second, to ensure our framework is effective across different classes of software weaknesses, we aligned our selection with the *2024 CWE Top 25 Most Dangerous Software Weaknesses* [30]. We queried the Open Source Vulnerability (OSV [56]) database to retrieve npm-specific vulnerabilities and classified them by CWE category. Categories lacking npm-specific entries (e.g., CWE-798 and CWE-306) were omitted. To ensure structural evaluability, we retained only vulnerabilities with at least 10 downstream dependents in our dataset (DS_{npm}). Within each category, we prioritized candidates by severity (Critical/High) and recency. After removing duplicates (e.g., CVE-2024-48914 covering multiple CWEs), this process yielded 22 distinct CVEs representing diverse weakness types such as Injection (CWE-79) and Prototype Pollution (CWE-1321).

For all candidates, we performed manual analysis to identify the precise vulnerable functions (Vul_{func}). We excluded CVE-2024-27094 (affecting `@openzeppelin/contracts`) as the vulnerability resides in Solidity smart contract code, outside the scope of our JavaScript analysis. Consequently, our final dataset comprises 27 valid CVEs, whose identified vulnerable functions serve as ground truth for the function-level reachability analysis in Section V.

TABLE VII: List of selected high-impact vulnerabilities. # Vul_{func} denotes the number of vulnerable functions.

CVE ID	Package Name	Downloads (2024)	# Vul_{func}
CVE-2021-23337	`lodash`	2.68B	1
CVE-2022-25883	`semver`	16.57B	14
CVE-2017-16137	`debug`	13.61B	1
CVE-2017-20165	`debug`	13.61B	1
CVE-2022-3517	`minimatch`	9.78B	7
CVE-2016-10540	`minimatch`	9.78B	5

TABLE VIII: Detailed list of selected vulnerabilities for diversity evaluation (CWE-Top-25). #Vul_{func} denotes the number of vulnerable functions.

CWE ID	Package Name	CVE ID	# Vul_{func}
CWE-79	happy-dom	CVE-2024-51757	4
CWE-787	electron	CVE-2022-4135	2
CWE-89	parse-server	CVE-2024-27298	4
CWE-352	whistle	CVE-2024-55500	5
CWE-22	@vendure/ asset-server-plugin	CVE-2024-48914	7
CWE-125	@openzeppelin/contracts	CVE-2024-27094	-
CWE-78	find-exec	CVE-2023-40582	3
CWE-416	@fastly/js-compute	CVE-2024-38375	7
CWE-862	snarkjs	CVE-2023-33252	3
CWE-434	strapi	CVE-2022-27263	3
CWE-94	angular-expressions	CVE-2024-54152	1
CWE-20	@vendure/ asset-server-plugin	CVE-2024-48914	7
CWE-77	openssl	CVE-2023-49210	1
CWE-287	isolated-vm	CVE-2022-39266	5
CWE-269	@aws-amplify/cli	CVE-2024-28056	4
CWE-502	gatsby-plugin-mdx	CVE-2022-25863	4
CWE-200	eventsource	CVE-2022-1650	1
CWE-863	next-auth	CVE-2022-35924	3
CWE-918	parse-url	CVE-2022-2900	1
CWE-119	@solana/web3.js	CVE-2024-30253	5
CWE-476	ws	CVE-2024-37890	2
CWE-798	-	-	-
CWE-190	@chainsafe/lodestar	CVE-2022-29219	1
CWE-400	@stryker-mutator/util	CVE-2024-57085	1
CWE-306	-	-	-

Algorithm 1 The compositional synthesis algorithm.

Input: P_{app}: The target application package; D: The set of dependencies.
Output: The resolved synthesis graph for P_{app}.
procedure SYNTHESIZEECG(P_{app}, D)
  $L \leftarrow$ ReverseTopologicalSort($D \cup \{P_{app}\}$)
  ResolvedGraphs $\leftarrow$ new Map()
  for all package P_i in L **do**
    $G_i \leftarrow$ GetOriginalRSG(P_i)
    for all dependency P_j of P_i **do**
      $G_j \leftarrow$ *ResolvedGraphs.get*(P_j)
      $G_i \leftarrow$ COMPOSE(G_i, G_j)
    end for
    ResolvedGraphs.put(P_i, G_i)
  end for
  return *ResolvedGraphs.get*(P_{app})
end procedure

function COMPOSE(G_{caller}, G_{callee})
  $G_{new} \leftarrow G_{caller} \cup G_{callee}$
  Let $\langle \mathcal{M}_{E,c}, \mathcal{M}_{I,c} \rangle \leftarrow$ GetContract(G_{caller})
  Let $\langle \mathcal{M}_{E,d}, \mathcal{M}_{I,d} \rangle \leftarrow$ GetContract(G_{callee})
  for all path $u \in$ domain($\mathcal{M}_{I,c}$) **do**
    if u targets package of G_{callee} **then**
      MATCHANDRESOLVE($u, \mathcal{M}_{E,d}, G_{new}$)
    end if
  end for
  return G_{new}
end function

TABLE IX: Comprehensive single-hop reachability analysis merging High-Impact and Diversity datasets. The global average is weighted based on the number of CVEs in each dataset.

Dimension	CVE ID	Package	$\#d_0$	$\#d_1$	$\#C_{\text{mod}}$	$\#C_{\text{func}}$	$\#C_{\text{vuln\_func}}$
High-Impact	CVE-2021-23337	lodash	100	396,112	264,179 (66.69%)	244,130 (61.63%)	11,574 (2.92%)
	CVE-2022-3517	minimatch	26	38,112	28,667 (75.22%)	15,791 (41.43%)	15,791 (41.43%)
	CVE-2016-10540	minimatch	23	10,341	9,211 (89.07%)	3,528 (34.12%)	3,528 (34.12%)
	CVE-2022-25883	semver	74	139,257	111,138 (79.81%)	102,209 (73.40%)	73,314 (52.65%)
	CVE-2017-16137	debug	55	70,297	54,098 (76.96%)	51,425 (73.15%)	50,454 (71.77%)
	CVE-2017-20165	debug	42	39,365	29,702 (75.45%)	29,583 (75.15%)	29,576 (75.13%)
Diversity	CVE-2022-1650	eventsource	17	167	109 (65.27%)	100 (59.88%)	100 (59.88%)
	CVE-2022-25863	gatsby-plugin-mdx	125	610	286 (46.89%)	0 (0.00%)	0 (0.00%)
	CVE-2022-27263	strapi	16	30	3 (10.00%)	3 (10.00%)	3 (10.00%)
	CVE-2022-2900	parse-url	11	204	67 (32.84%)	63 (30.88%)	63 (30.88%)
	CVE-2022-29219	@chainsafe/lodestar	23	23	17 (73.91%)	11 (47.83%)	0 (0.00%)
	CVE-2022-35924	next-auth	17	58	34 (58.62%)	10 (17.24%)	9 (15.52%)
	CVE-2022-39266	isolated-vm	15	38	25 (65.79%)	25 (65.79%)	25 (65.79%)
	CVE-2022-4135	electron	504	2,453	1,978 (80.64%)	1,816 (74.03%)	1,816 (74.03%)
	CVE-2023-33252	snarkjs	27	309	243 (78.64%)	220 (71.20%)	148 (47.90%)
	CVE-2023-40582	find-exec	8	11	11 (100.00%)	11 (100.00%)	11 (100.00%)
	CVE-2023-49210	openssl	2	56	0 (0.00%)	0 (0.00%)	0 (0.00%)
	CVE-2024-27298	parse-server	9	27	13 (48.15%)	8 (29.63%)	8 (29.63%)
	CVE-2024-28056	@aws-amplify/cli	5	13	0 (0.00%)	0 (0.00%)	0 (0.00%)
	CVE-2024-30253	@solana/web3.js	109	428	400 (93.46%)	381 (89.02%)	168 (39.25%)
	CVE-2024-37890	ws	86	4,080	3,163 (77.52%)	2,389 (58.55%)	1,561 (38.26%)
	CVE-2024-38375	@fastly/js-compute	24	24	0 (0.00%)	0 (0.00%)	0 (0.00%)
	CVE-2024-48914	@vendure/asset-server-plugin	38	41	0 (0.00%)	5 (12.20%)	5 (12.20%)
	CVE-2024-51757	happy-dom	150	452	252 (55.75%)	257 (56.86%)	13 (2.88%)
	CVE-2024-54152	angular-expressions	8	65	44 (67.69%)	42 (64.62%)	41 (63.08%)
	CVE-2024-55500	whistle	7	27	13 (48.15%)	13 (48.15%)	0 (0.00%)
	CVE-2024-57085	@stryker-mutator/util	80	640	588 (91.88%)	546 (85.31%)	64 (10.00%)
Average			-	-	**57.72%**	**45.71%**	**32.49%**

TABLE X: Comparative analysis of vulnerability propagation at package and function levels (Multi-hop). The table merges High-Impact and Diversity datasets. **RP** denotes the relative proportion of Function-level to Package-level.

Dimension	CVE ID	Paths	Affected Library			Affected Version			Avg. Hop Distance		Max Hop Distance	
			#P-L	#F-L	RP	#P-L	#F-L	RP	P-L	F-L	P-L	F-L
High-Impact	CVE-2021-23337	285,622	17,267	4,826	27.95%	166,554	37,220	22.35%	3.34	0.61	13	6
	CVE-2022-3517	497,595	21,775	1,211	5.56%	286,731	22,557	7.87%	4.15	0.12	13	5
	CVE-2016-10540	127,976	14,499	649	4.48%	84,886	7,916	9.33%	3.65	1.14	12	5
	CVE-2022-25883	2,096,181	48,192	3,863	8.02%	595,078	74,823	12.57%	7.48	1.09	32	7
	CVE-2017-16137	6,663,049	50,444	23,107	45.81%	711,199	286,679	40.31%	7.39	2.05	16	8
	CVE-2017-20165	6,184,586	39,460	18,729	47.46%	488,064	208,227	42.66%	7.62	2.08	15	8
Diversity	CVE-2022-1650	737	750	92	12.27%	802	120	14.96%	2.49	0.88	6	2
	CVE-2022-25863	810	620	81	13.06%	1,022	125	12.23%	1.30	0.00	4	0
	CVE-2022-27263	30	26	2	7.69%	46	19	41.30%	0.65	0.16	1	1
	CVE-2022-2900	538	516	154	29.84%	551	178	32.30%	2.60	2.47	7	6
	CVE-2022-29219	23	2	2	100.00%	46	24	52.17%	0.50	0.04	1	1
	CVE-2022-35924	67	45	10	22.22%	82	27	32.93%	0.88	0.37	2	1
	CVE-2022-39266	47	37	15	40.54%	74	41	55.41%	1.18	0.66	4	2
	CVE-2022-4135	6,832	1,686	1,037	61.51%	3,604	2,528	70.14%	1.07	0.84	5	4
	CVE-2023-33252	1,011	405	129	31.85%	567	202	35.63%	1.59	1.01	7	4
	CVE-2023-40582	16	14	12	85.71%	29	27	93.10%	1.07	1.00	2	2
	CVE-2023-49210	56	56	1	1.79%	58	2	3.45%	0.97	0.00	1	0
	CVE-2024-27298	27	24	7	29.17%	37	17	45.95%	0.78	0.47	2	1
	CVE-2024-28056	13	13	1	7.69%	19	5	26.32%	0.79	0.00	2	0
	CVE-2024-30253	623	460	374	81.30%	709	596	84.06%	1.14	1.01	4	4
	CVE-2024-37890	309,912	16,852	2,298	13.64%	20,796	3,108	14.95%	2.75	1.36	13	5
	CVE-2024-38375	24	2	1	50.00%	48	24	50.00%	0.50	0.00	1	0
	CVE-2024-48914	3,783	56	2	3.57%	880	43	4.89%	1.88	0.12	3	1
	CVE-2024-51757	760	284	32	11.27%	877	247	28.16%	1.21	0.42	4	2
	CVE-2024-54152	69	71	42	59.15%	82	51	62.20%	1.01	0.88	2	2
	CVE-2024-55500	27	21	8	38.10%	34	19	55.88%	0.79	0.63	1	1
	CVE-2024-57085	662	42	7	16.67%	759	144	18.97%	0.95	0.44	3	1
Global Average			-	-	**31.72%**	-	-	**34.74%**	**2.21**	**0.71**	-	-

P-L indicates package-level analysis, **F-L** indicates function-level analysis, and **RP** represents the relative proportion of **F-L** to **P-L**.

TABLE XI: The comprehensive list of selected packages in the performance evaluation.

#	Root Package ID	# Dep.	kLOC	# Vul.
1	@xapp/stentor-templates@1.22.0	305	1,741	1
2	@xapp/stentor-service-analytics@1.25.4	310	1,623	4
3	@graphback/codegen-client@0.16.1	105	163	1
4	stentor-handler@1.45.5	143	990	4
5	stentor-conditional@1.33.62	114	906	1
6	@accordproject/cicero-test@0.11.2-20190401205913	484	434	2
7	stentor-handler-delegating@1.27.11	209	1,489	4
8	stentor-conditional@1.57.42	119	940	1
9	@xapp/stentor-service-sms@1.31.24	156	1,014	1
10	@accordproject/ergo-engine@0.8.1-20190426004645	140	115	1
11	@xapp/stentor-handler-sms@1.20.10	304	1,715	8
12	stentor-response@1.29.13	205	1,454	1
13	@accordproject/cicero-cli@0.20.11-20200410183245	506	562	7
14	redstone-smartweave@0.4.83	116	165	1
15	stentor-determiner@1.56.37	137	1,024	1
16	@xapp/stentor-actions-on-google@1.32.13	205	1,463	1
17	@jsenv/plugin-commonjs@0.0.14	73	136	1
18	@graphql-cli/serve@4.1.1-alpha-b6d19c4.0	580	1,351	24
19	stentor-runtime@1.57.39	155	1,040	31
20	@accordproject/ergo-engine@0.8.5-20190608110624	140	115	1
21	scriptworks-client@0.10.16	789	1,197	1
22	@xapp/stentor-service@1.26.14	221	1,468	1
23	stentor-runtime@1.29.0	217	1,454	31
24	jsreport-core@2.10.0	93	107	1
25	stentor-determiner@1.36.17	132	1,002	1
26	stentor-handler@1.56.16	145	1,024	4
27	@xapp/stentor-handler-media@1.32.68	259	1,677	8
28	@xapp/stentor-alexa@1.26.43	157	1,044	1
29	@xapp/stentor-lex-v2@1.40.313	328	1,638	3
30	@serverless-toolkit/stacks@3.0.11	126	3,351	1
31	@xapp/stentor-service-sms@1.25.2	222	1,462	1
32	stentor-context@1.57.8	143	1,033	1
33	@xapp/stentor-media-manager@1.29.24	188	1,206	1
34	@xapp/stentor-service@1.26.12	221	1,468	1
35	@xapp/stentor-dialogflow-facebook@1.18.147	319	1,721	3
36	azupck@1.2.29	1,286	1,632	10
37	stentor-determiner@1.33.86	134	1,002	1
38	@xapp/stentor-templates@1.24.52	258	1,676	1
39	@accordproject/ergo-engine@0.8.1-20190426121823	140	115	1
40	@xapp/stentor-service-smapi@1.26.58	166	1,048	2
41	stentor-determiner@1.26.8	186	1,065	1
42	@graphql-cli/generate@4.0.1-alpha-cf8155c.0	281	298	4
43	@graphql-cli/serve@4.1.1-alpha-d207ff9.0	559	626	24
44	@xapp/stentor-lex-v2@1.40.38	343	1,718	3
45	@accordproject/cicero-cli@0.10.1-20190122220137	466	384	1
46	@xapp/stentor-media-manager@1.31.16	191	1,206	1
47	stentor-response@1.27.46	205	1,468	1
48	@xapp/stentor-service-analytics@1.19.5	355	1,694	4
49	@accordproject/ergo-cli@0.8.6-20190620190000	262	347	2
50	@accordproject/cicero-server@0.4.4-20180615051811	430	321	1
51	stentor-handler-factory@1.26.49	211	1,488	10
52	@xapp/stentor-service-analytics@1.26.65	308	1,622	4
53	@accordproject/cicero-test@0.21.27-20210319042108	325	538	4
54	stentor-handler-delegating@1.45.13	144	990	4
55	stentor-handler-delegating@1.42.14	144	979	4
56	@xapp/stentor-dialogflow@1.32.50	263	1,592	2
57	@xapp/stentor-alexa@1.22.1	221	1,517	1
58	@accordproject/ergo-engine@0.0.64-20180611191608	157	243	1
59	@xapp/stentor-templates@1.18.112	303	1,666	2
60	stentor-determiner@1.27.2	137	1,028	1
61	stentor-handler-factory@1.33.31	208	1,462	10
62	@xapp/stentor-migration@1.32.45	265	1,600	3
63	@xapp/stentor-service-lex@1.40.22	289	1,169	4
64	@xapp/stentor-service-smapi@1.32.9	169	1,048	2
65	stentor-context@1.42.41	141	988	1
66	@xapp/stentor-alexa@1.29.20	153	1,044	1
67	@xapp/stentor-alexa@1.32.25	153	1,044	1
68	stentor@1.55.19	221	1,064	48
69	@accordproject/cicero-cli@0.13.5-20190914214832	486	463	3
70	@graphql-cli/serve@4.1.1-alpha-b67f6f3.0	580	1,352	24
71	@xapp/stentor-service-smapi@1.28.7	165	1,048	2
72	@graphql-cli/serve@4.1.1-alpha-0ef0ed7.0	568	1,338	24
73	@jsenv/plugin-react@1.0.1	250	553	1
74	youtube-studio@0.0.21	41	69	1
75	@xapp/stentor-handler-sms@1.32.99	262	1,693	16
76	@accordproject/cicero-server@0.22.0-20210413122815	305	432	3
77	@xapp/stentor-service-sms@1.32.29	157	1,014	1
78	@xapp/stentor-interaction-model-profiler@1.24.78	293	1,620	5
79	stentor-determiner@1.23.55	184	1,058	1
80	stentor-handler-manager@1.57.18	148	1,034	15
81	stentor-context@1.57.1	144	1,034	1
82	stentor-handler-delegating@1.37.22	141	1,005	4
83	@graphback/codegen-schema@0.16.0-beta3	105	164	2
84	@graphql-cli/generate@4.0.1-alpha-db7cff3.0	280	329	4
85	@accordproject/cicero-server@0.20.0-20191029124228	413	332	3
86	stentor-handler-factory@1.45.19	145	993	10
87	stentor@1.36.35	216	1,041	48
88	@xapp/stentor-templates@1.31.14	326	1,813	6
89	@xapp/stentor-lex-connect@1.26.70	209	1,471	1
90	@accordproject/ergo-cli@0.1.3-20180730181229	152	256	1
91	stentor-context@1.26.37	256	1,551	1
92	@accordproject/cicero-test@0.11.2-20190326183124	491	429	2
93	@graphql-cli/serve@4.1.1-alpha-16c9eb3.0	560	1,355	24
94	stentor@1.53.6	219	1,054	48
95	@xapp/stentor-media-manager@1.27.4	190	1,203	1
96	@graphql-cli/generate@4.1.1-alpha-36fdb45.0	306	1,095	4
97	@dfeidao/server@4.6.201909041441	655	660	1
98	@akashic/akashic-cli@2.15.56	753	967	1
99	@akashic/akashic-cli-serve@1.14.65	513	844	1

Appendix A
Artifact Appendix

A. Description & Requirements

In this work, we propose a novel framework VulTracer, which can precisely and efficiently perform vulnerability propagation analysis at function level. The artifacts implements the framework's three core components: Rich Semantic Graph Generation, Interface Contract Extraction, and Compositional Synthesis. We also provide a minimal set of dependencies and sample packages to demonstrate the end-to-end workflow.

1) How to access: The artifact is accessible via Zenodo.[2]

2) Hardware dependencies:

- **CPU:** at least 2 cores.
- **RAM:** at least 16 GB.
- **Disk:** at least 8 GB free.

3) Software dependencies:

- **System:** 64-bit Linux, macOS or Windows (WSL2).
- **Docker:** 20.10 or newer.

4) Benchmarks: None.

B. Artifact Installation & Configuration

First, download the files from Zenodo as described above. Because Zenodo provides the materials as compressed archives, extract them before proceeding. Then follow the steps in the main repository's README. We designed the artifacts with cross-platform reproducibility in mind. The code runs in a Dockerized environment and can be executed on any system with Docker installed. Building the container prepares all dependencies. Once built successfully, simply run the test scripts referenced in the README to complete the AEC evaluation.

[2]https://doi.org/10.5281/zenodo.17659795